Is Cloned Code more stable than Non-Cloned Code?

Jens Krinke
FernUniversität in Hagen
Hagen, Germany

Abstract

This paper presents a study on the stability of cloned code. The results from an analysis of 200 weeks of evolution of five software system show that the stability as measured by changes to the system is dominated by the deletion of code clones. It can also be observed that additions to a systems are more often additions to non-cloned code than additions to cloned code. If the dominating factor of deletions is eliminated, it can generally be concluded that cloned code is more stable than non-cloned code.

1. Introduction

The duplication of code is common practice to make software development faster, to enable "experimental" development without impacting the original code, or to enable independent evolution [8]. Since these practices involve both duplication and modification, they are collectively called *code cloning*. While the duplicated code is called a *code clone*. A *clone group* consists of code clones that are clones of each other (sometimes this is also called a *clone class*). During the software development cycle, code cloning is easy and inexpensive (in both effort and money). However, this practice can complicate software maintenance and it has been suggested that too much cloned code is a risk, albeit the practice itself is not generally harmful [13]. Because of these problems, many approaches to detect cloned code have been developed [5,6,9,12,16–18,21]. Empirical studies of cloned code have focused mainly on examining whether code clones are changed consistently [4, 10, 15, 19].

Another important question is if cloned code is more stable than non-cloned code during the evolution of a system, i.e. if non-cloned code is changed more often than cloned code. If cloned code is generally less stable than non-cloned code, it can be assumed that cloned code requires more attention and is indeed more expensive to maintain. If cloned code is generally more stable than non-cloned code, its maintenance costs will be lower.

We sought to answer this question by studying the changes that are applied to cloned and non-cloned code during 200 weeks of evolution of five open source software systems. For the cases studied, the following facts were found:

- The amount of deletions that occur during the evolution of a system is dominating, i.e. there exist massive deletions of cloned code.
- The average percentage of additions, deletions, or other changes to cloned code is lower than the average percentage for non-cloned code.
- It is more often the case that a higher percentage of non-cloned code is added, deleted, or changed in comparison to cloned code.
- Non-cloned code is more often changed (incl. additions and deletions) than cloned code and therefore, cloned is more stable than non-cloned-code.

The next section presents the theoretical framework that defines changes, code clones, and clone groups. The setup of the empirical study is presented in Section 3 and its results in Section 4. After Section 5 discusses related work, the last section will conclude.

2. A Framework for Changes to Clones

This section will present the framework in which code clones, groups of code clones, and changes to code clones are defined and related to the evolution of software systems in terms of the versions of the system.

2.1. Code Clones

Code clones are usually described as source code ranges (or fragments) that are identical or very similar. They are grouped into *clone groups* (sometime called *clone classes*) which are sets of identical or very similar code clones. A code clone $c = (s, l, f)$ is the source code range starting at line s with the following l lines of code in file f, thus the last line of the code clone is $s + l - 1$. A clone group $G =$

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.14

$\{c_1, \ldots, c_n\}$ is a set of n code clones $c_1, \ldots, c_n$, where each of the code clones is a clone of the others. For the purpose of this study, the effects of *split* or *fragmented* code clones are ignored. Such clones would consist of multiple source code ranges in the same file. An example of such a code clone is a source code range that is copied and afterward additional source code is inserted into the cloned code.

The code clones do not have to be disjoint: it is possible for two code clones $c_1 = (s_1, l_1, f)$ and $c_2 = (s_2, l_2, f)$ to share a common source range ($min(s_1 + l_1, s_2 + l_2) > max(s_1, s_2)$).

Most of the available tools for code clone detection generate a list of clone groups. Usually, a minimal size k of an identified code clone $c = (s, l, f)$ can be specified, i.e. $size(c) > k$. In the following it is assumed that $size(c) = l$. However, many tools use the number of lexical tokens that is covered by a code clone as the size of the code clone.

2.2. Changes

Source code changes are frequently defined in terms of a source code range that has been replaced by other source code. A change $d = (s, l, f, n, t)$ is the source code range starting at line s in file f with a size $l \geq 0$ (number of lines) that will be replaced by the text t with a size of $n \geq 0$ (number of lines). Note that a change is not specified with the last line number because that would not allow the specification of an empty range. The kind of change is usually distinguished between *deletion*, *addition*, or *change*. Addition and deletion are special cases with $l = 0$ or $n = 0$, respectively. The changes to a program can be expressed by a set of changes $D = \{d_1, \ldots, d_k\}$ where the d_i do not overlap.

2.3. Software Systems and their Version History

For the purpose of the study, a software system $S = \{f_1, \ldots, f_n\}$ consists of a set of n source code files $f_i, 1 \leq i \leq n$. Independent of the specific kind of version, it is assumed that a system exists in multiple versions $v \in V$ where the complete system can be retrieved for every version v: $S(v) = \{f_1^v, \ldots, f_n^v\}$ is the system in version v. In a versioning system like CVS, the version v can be specified in a very flexible way. For example it can be specified as a time or by a name given to a specific version (branching will be ignored in this work). The differences between two versions x and y of a system can be identified by a set of changes as described above: Let $D(v, w)$ be the set of changes $\{d_1, \ldots, d_k\}$ between $S(v)$ and $S(w)$.

2.4. Changes to Cloned and Non-Cloned Code

In this study, the changes to a system S between two versions v and w will be analyzed with respect to the clones in the system. As the ultimate goal is to measure the stability of cloned and non-cloned code, we need a measure for comparing, between subsequent versions of a system, the amount of changes to cloned versus non-cloned code.

Let $c(v)$ be the set of code clones $\{c_1, \ldots, c_n\}$ in $S(v)$. For any source code line in the system $S(v)$ it is possible to say if it is *cloned* or *non-cloned*. A source line s in file f is cloned, iff $\exists c_1 \in c(v) : c_1 = (s_1, l_1, f) \wedge s_1 \leq s < s_1 + l_1$. Note that clones are allowed to overlap and there may exist more than one c_1. Let $C(v)$ be the set of cloned lines and $N(v)$ the set of non-cloned lines in version v.

For any source line in the system $S(v)$ it is also possible to say if it is *changed* or *unchanged* between versions v and w. A source line s in file f is changed, iff $\exists d \in D(v, w) : d = (s_1, l_1, f, n, t) \wedge s_1 \leq s < s_1 + l_1$. Note that this only covers changes and deletions, but not additions (because of $l_1 = 0$). Therefore it is necessary to handle changes, deletions, and additions independently:

- A source line s in file f gets deleted if in addition to the above condition $n = 0$ holds. Let $DC(v)$ be the set of source lines that get deleted and are cloned according to the above condition in version v. Let $DN(v)$ be the corresponding set of non-cloned deleted lines.
- A source line s in file f gets changed if additionally $n > 0$ holds. Let $CC(v)$ be the set of cloned changed lines and $CN(v)$ the set of non-cloned changed lines.
- A (new) source line s in file f gets added if the condition $\exists d_1 \in D(v, w) : d = (s_1, 0, f, n, t) \wedge s_1 \leq s < s_1 + n$ holds instead of the above. It is not possible to distinguish if the added line is cloned or non-cloned, however, it is possible to distinguish if it gets added to cloned or non-cloned code: Let $d = (s_1, 0, f, n, t)$ be the addition. The n added lines are added to cloned code, iff $\exists c_2 \in c(v) : c_2 = (s_2, l_2, f) \wedge s_2 < s_1 < s_2 + l_2$. Note that it is impossible to distinguish an addition before or to a clone for $s_1 = s_2$, thus it is assumed that additions with $s_1 = s_2$ are always additions before the clone and do not belong to the cloned code. Let $AC(v)$ be the set of lines that are added to cloned code and let $AN(v)$ be the set of lines that are added to non-cloned code.

2.5. Measuring Stability

With the now defined framework it is possible to measure the amount of changes within cloned and non-cloned code. Because additions, deletions, and changes are operations with different effects on cloned and non-cloned code

(as we will see), the measurements will distinguish the three possible operations:

- The *instability* in respect to deletions is for cloned code $\frac{\sum_{v \in V} |DC(v)|}{\sum_{v \in V} |C(v)|}$ and for non-cloned code $\frac{\sum_{v \in V} |DN(v)|}{\sum_{v \in V} |N(v)|}$.
- The *instability* in respect to additions is for cloned code $\frac{\sum_{v \in V} |AC(v)|}{\sum_{v \in V} |C(v)|}$ and for non-cloned code $\frac{\sum_{v \in V} |AN(v)|}{\sum_{v \in V} |N(v)|}$.
- The *instability* in respect to changes is for cloned code $\frac{\sum_{v \in V} |AC(v)|}{\sum_{v \in V} |C(v)|}$ and for non-cloned code $\frac{\sum_{v \in V} |AN(v)|}{\sum_{v \in V} |N(v)|}$.

The instability is basically defined in respect to the number of deleted, added, or changed lines in comparison to all lines. To be able to clearly state that cloned code in general is more stable than non-cloned code, the instability measures should all be much lower for the cloned code than for the non-cloned code. The following experiment will therefore measure the instability of five software systems.

3. Experiment Setup

For the study the version histories of five open source systems have been retrieved. All five systems have to have a sufficiently long development history, which is the case if the system has reached a released state before 2002-08-08 and has been in further development since then. The five systems are:

1. ArgoUML[1] is a UML modeling tool that includes support for standard UML diagrams. It is written in Java and its version archive is available via subversion.
2. The JDT core subsystem of Eclipse[2]: From Eclipse's version archive the `org.eclipse.jdt.core` module has been used.
3. GNU Emacs[3] is the famous text editor, written in C.
4. FileZilla[4] is a FTP client with a graphical user interface for Windows, written in C++.
5. SQuirreL[5] is a graphical SQL client written in Java.

All five systems are large enough (¿50KLOC) and have enough changes (¿500 changed lines per week on average) in their version archive within the 200 observed weeks. Moreover, they cover different applications, different platforms, and different programming languages. One of the systems, ArgoUML, has been used in previous studies by Kim et al. [15] and Aversano et al. [4]. The other systems used by the studies have not been used because they had too few and too small changes affecting clones and thus have been rejected. Four of the five systems have also been used in a study on consistent and inconsistent changes to clones [19].

System	Source LOC	Cloned LOC	
ArgoUML	118316	14335	12%
jdt.core	192624	28149	15%
Emacs	227919	21840	10%
FileZilla	90302	14060	16%
SQuirreL	69981	5773	8%

Table 1. Analyzed systems

The sources of all five systems have been retrieved based on their status in the version archive on 200 different dates, such that each version is exactly one week later or earlier than the next or previous version. A one week cycle has been chosen because CVS activity is usually dependent on the weekday [23] and projects often use a week oriented process (e.g. within Eclipse). For all systems, the first version was from 2002-08-08 and the last version was from 2006-06-01.

In all systems, only the Java, C, and C++ source and header files have been analyzed. Also, the source files have been transformed to eliminate spurious changes between versions: Comments were removed from the sources and afterward the source files have been reformatted with the pretty printer *Artistic Style*[6]. The transformed sources are saved to a repository. With this repository, all $S(v), 0 \leq v \leq 200$ can be accessed.

The changes between the versions of the systems have been identified by the standard *diff* tool. For each version v of the analyzed system, the changes between version v and $v + 1$ (the version of the next week) have been identified, generating $D(v, v + 1)$.

For each of the 200 versions, the clone groups $G(v)$ have been identified by the use of the clone detection tool *Simian*[7] from RedHill Consulting Pty. Ltd. It is a text-based clone detector that detects almost identical clones. Most of the other (freely) available clone detectors cannot be used because they require the use of a GUI or are restricted to Java source files. *Simian* has been instructed to identify clones with a size of at least 11 source code lines. The possibility to relax the identification by assuming that all literals are identical has not been used.

[1] `http://argouml.tigris.org/`
[2] `http://www.eclipse.org/`
[3] `http://www.gnu.org/software/emacs/`
[4] `http://filezilla.sourceforge.net/`
[5] `http://squirrel-sql.sourceforge.net/`
[6] `http://astyle.sourceforge.net/`
[7] Available at `http://www.redhillconsulting.com.au/products/simian/index.html`

The framework described in the previous section has been implemented in a tool that takes a list of clone groups $G(v)$ as detected by *Simian* and a list of changes $D(v, w)$ as produced by *diff* that are then mapped on the code clones. The tool will compute the needed sets as described in the previous section.

The analysis has been done on 200 versions. For each week w, $1 \leq w \leq 200$, the tool has generated the needed values for $|C(w)|$, $|N(w)|$, $|AC(w)|$, $|AN(w)|$, $|DC(w)|$, $|DN(w)|$, $|CC(w)|$, and $|CN(w)|$ based on the clone groups of the analyzed system in week $w-1$ and the changes from week $w-1$ to week w.

Table 1 shows some properties of the analyzed systems: The second column contains the average size of the analyzed source base (in LOC) for a week. The next two columns contain the average size of cloned source code (in LOC and as an percentage of the source code). For example, GNU Emacs is the largest system with 228 KLOC on average, from which 10% (22 KLOC) is cloned code on average.

4. Results

This section presents the results of the study as described in the previous section. Figure 1 shows the instability measures for all five systems. The grey bars show the instability for additions ("AC%"), deletions ("DC%"), and changes ("CC%") for cloned code and the white bars show the same for non-cloned code ("AN%", "DN%", and "CN%"). From this figure, the following conclusions can be drawn:

- Less is added to cloned code than to non-cloned code for all five systems. For example, in ArgoUML only 0.07% is added to cloned code on average in comparison to 0.27% of additions to non-cloned code. In respect to additions, cloned code is more stable than non-cloned code.
- From cloned code is more deleted than from non-cloned code. For example in ArgoUML, 0.85% of the cloned code is deleted on average in comparison to 0.63% of the non-cloned code. It seems that in respect to deletions, non-cloned code is more stable than cloned code.
- For changes, the picture is not clear: In three out of five systems, cloned code is more stable in respect to changes than non-cloned code.

Remember that the higher percentages of deletions in comparison to the additions do not indicate shrinking system sizes because of the differences in measuring the changes, additions, and deletions. To better understand the patterns, two of the systems will be looked at in detail.

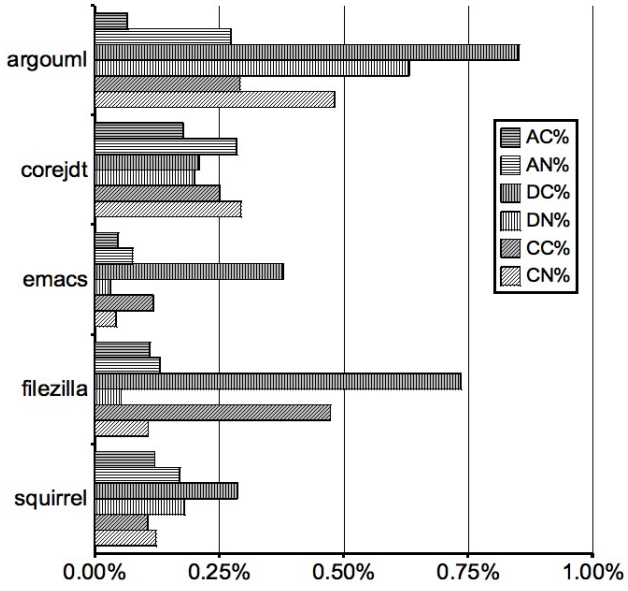

Figure 1. Summary of the results

4.1. ArgoUML

Figure 2 shows the system size of ArgoUML measured in kLOC for 200 weeks. Overall, the size increases along the time except for two time periods: There is a huge drop in size around week 85 and there is a temporary drop in size during the weeks 185–187. The other variations in size are not significant. Figure 3 shows the rate of cloned code during the same 200 weeks ($\frac{|C(v)|}{|N(v)+C(v)|}$). The drop in size around week 85 has a clear correspondent drop in the amount of cloned code: In the same week, the amount of cloned code is almost halved. Until the drop, the rate of cloned code is decreasing and after the drop it stays almost stable around 8%.

The huge drop from week 84 to 85 can be explained by the operations performed on the software repository: During this week, six files have been removed from the repository which are two almost identical instances of a generated Java parser and lexer. These six files were responsible for almost half of the cloned code in the project. A similar event is responsible for the drop from week 184 to 185: During this week, another set of generated files for a class-file parser is deleted. As this set is a single instance, it has no significant impact on the number of clones.

Even now that it is clear that this restructuring with the massive deletion of clones is dominating the overall stability, it is still important to have a closer look at the evolution. Figure 4 compares the deletions in cloned and non-cloned code. The grey bars show the amount of deletions for cloned and non-cloned code as percentages. Positive values repre-

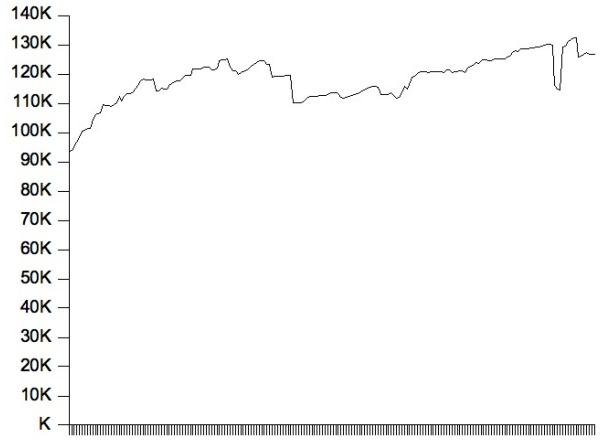

Figure 2. ArgoUML's size during 200 weeks

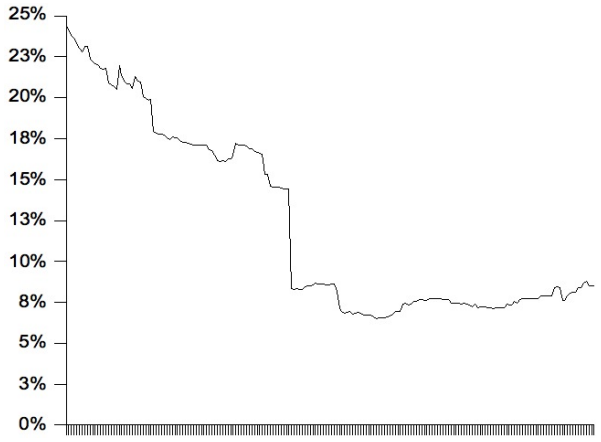

Figure 3. ArgoUML: Rate of cloned code

sent the rate of cloned code that gets deleted and the negative values represent the rate of non-cloned code. The rates are computed by $\frac{|DC(v)|}{|C(v)|}$ and $\frac{|DN(v)|}{|N(v)|}$ for every week v. For example, in week 85, 45% of the cloned code is deleted and 3.9% of the non cloned code (shown as -3.9%). The black line shows the difference between the two rates (as computed by $\frac{|DC(v)|}{|C(v)|} - \frac{|DN(v)|}{|N(v)|}$. This figure shows that there are large deletions of cloned code in weeks 33, 75, 77, 85, and 104. Large deletions of non-cloned code are in weeks 118, 127, 184, and 193. It seems that the large deletion of non-cloned code (almost 12%) in week 184 leads to the temporary drop in size mentioned above. It should also be noted that in 137 weeks the rate of deleted non-cloned code is larger than the rate of deleted cloned code ($\frac{|DC(v)|}{|C(v)|} < \frac{|DN(v)|}{|N(v)|}$) and in only 53 weeks it is the other way. Thus, usually during the evolution of ArgoUML, more non-cloned code gets deleted than cloned code.

Figure 5 shows the same diagram for the additions. As expected, there is a single week (166) where more than 1% of clone code is added to the cloned code. For the non-cloned code, this happens at week 2, 3, 5, 10, 39, 129, and 150. The black line already indicates the trend that usually more is added to the non-cloned code: $\frac{|AC(v)|}{|C(v)|} < \frac{|AN(v)|}{|N(v)|}$ holds in 174 weeks. All this indicates that for additions, cloned code is usually more stable than non-cloned code.

Figure 6 shows the diagram for changes. The pattern here is not as clear as in the other two discussed before. It seems that often if a large rate of cloned code is changed, a similar large rate of non-cloned code is changed, too. For example, in week 130, 4.6% of the cloned code is changed and 5.6% of the non-cloned code is changed. Again, usually the rate of changes to non-cloned code is larger than the rate for cloned code (in 145 weeks).

4.2. SQuirreL

Figure 7 shows the system size of SQuirreL for the 200 weeks. Overall, the size is increasing along the time and more than doubles during the time period. There is only one larger drop in size during week 126. Figure 8 shows the percentage of cloned code during the same 200 weeks: it changes between 5.3% and 9.8%. Large changes occur during week 28, 62, and 179 (increases) and during week 125 and 183 (decreases). The reasons for the changes are:

- In week 28 a new plugin for syntax highlighting has been added which has a lot of cloned code together with a similar plugin for jEdit. Indeed, the new plugin is supposed to replace the older plugin, but it was not removed until week 125.
- In week 62 a set of highly similar class files was added which represents different data types (e.g. "DataTypeFloat" and "DataTypeDouble"). The added code is identified as cloned in following weeks.
- In week 179 a set of five files for database schema have been added that are almost identical to a previous set in a different directory. The previous set of files is removed in week 183.

Again it is clear that large scale restructurings are dominating the changes in percentage of cloned code in SQuirreL. Figures 9, 10, and 11 show the trends for deletions, additions, and other changes for SQuirreL. The described events in week 125 and 183 can be clearly seen as peaks in Figure 9. The peaks in the first week are due to a restructuring of the jEdit plugin. The peaks in weeks 89 and 90 come from a massive change to the client code with a lot of added and deleted files.

It should also be noted that only in 29 weeks the percentage of deleted cloned code is larger than the percentage

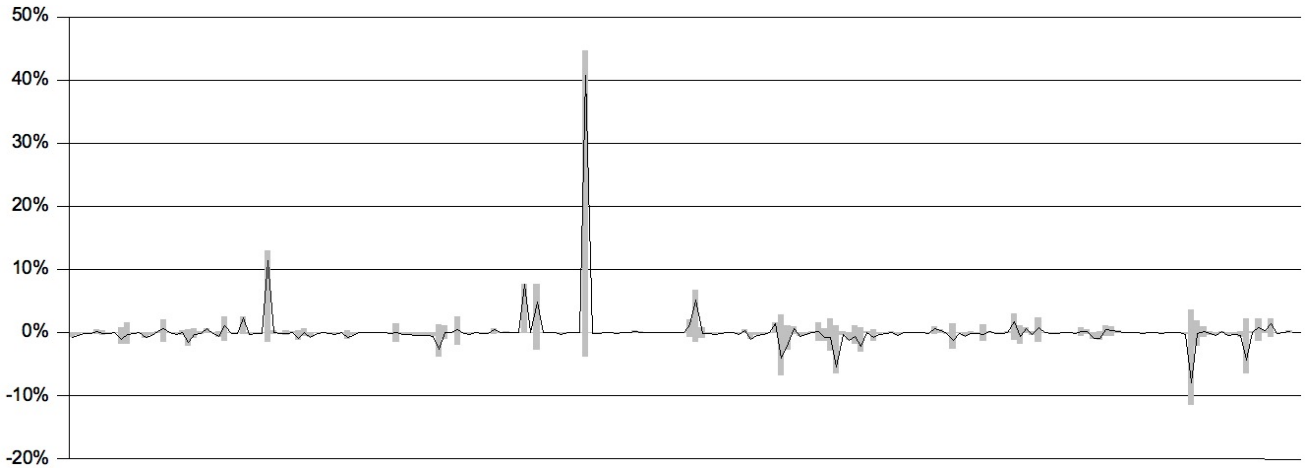

Figure 4. ArgoUML: Deletions

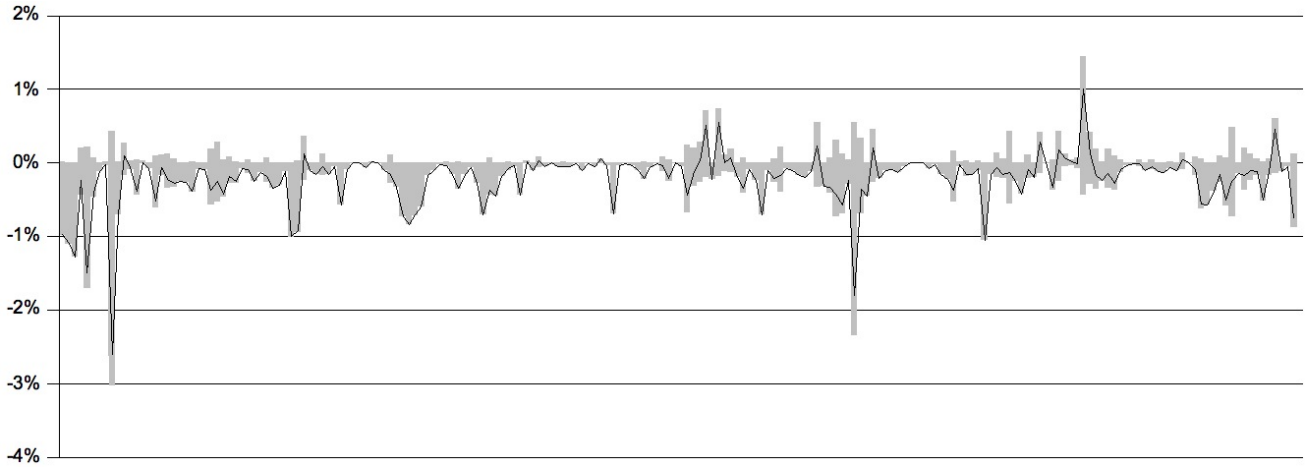

Figure 5. ArgoUML: Additions

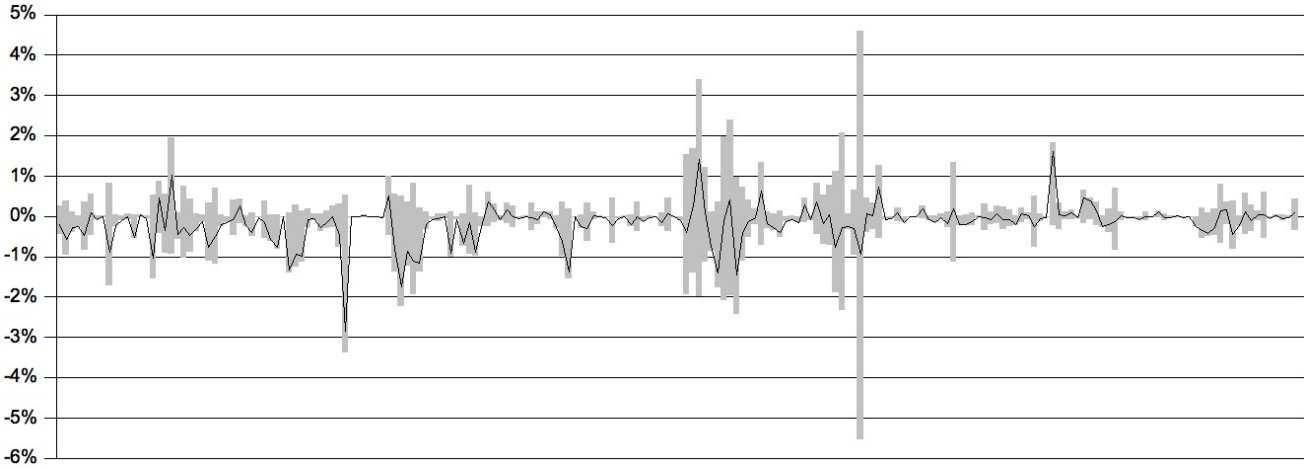

Figure 6. ArgoUML: Changes (that are neither Additions nor Deletions)

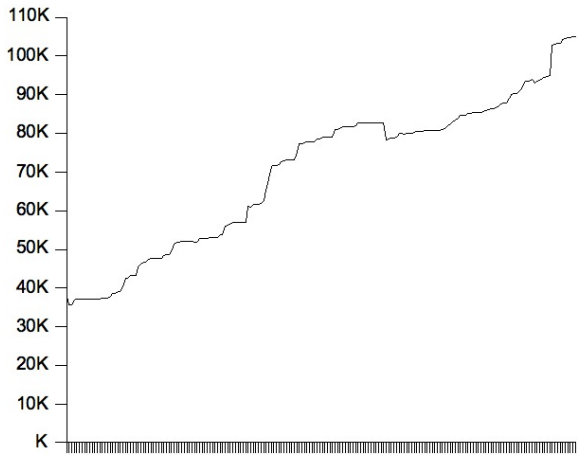

Figure 7. SQuirreL's size during 200 weeks

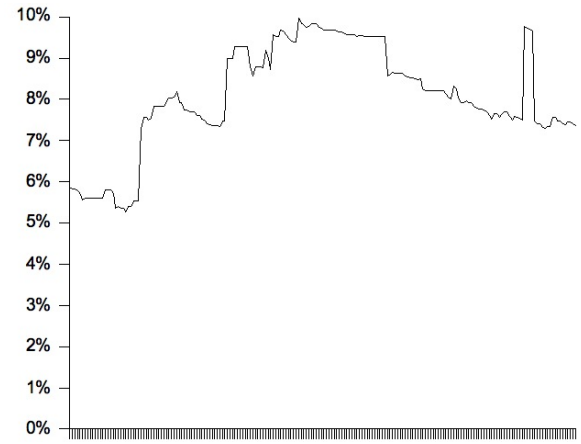

Figure 8. SQuirreL: Rate of cloned code

of deleted non-cloned code and in 106 weeks it is the other way. Thus, usually during the evolution of SQuirreL, more non-cloned code gets deleted than cloned code.

The peak in week 77 in Figure 10 comes from a massive addition to one of the data type representations: The string type gets a dedicated GUI during that week. However, the trend that usually more is added to the non-cloned code can be observed in 130 weeks and in 25 weeks there are more additions to cloned code. This indicates that for additions, cloned code is usually more stable than non-cloned code.

Figure 11 shows the same diagram for the changes. The pattern here is again not as clear as in the other two discussed before. There is one large peak in week 139 which is due to a massive refactoring and code cleanup as it seems. Again, usually the percentage of changes to non-cloned code is larger than the percentage for cloned code (in 120 weeks, 33 weeks it is vice versa).

	A	B	C	D	E	F
ArgoUML	174	21	137	53	145	52
jdt.core	155	37	127	60	140	55
Emacs	160	34	145	35	142	54
FileZilla	93	29	74	27	90	37
SQuirreL	130	25	106	29	120	33

A $\frac{|AC(v)|}{|C(v)|} < \frac{|AN(v)|}{|N(v)|}$ B $\frac{|AC(v)|}{|C(v)|} > \frac{|AN(v)|}{|N(v)|}$

C $\frac{|DC(v)|}{|C(v)|} < \frac{|DN(v)|}{|N(v)|}$ D $\frac{|DC(v)|}{|C(v)|} > \frac{|DN(v)|}{|N(v)|}$

E $\frac{|CC(v)|}{|C(v)|} < \frac{|CN(v)|}{|N(v)|}$ F $\frac{|CC(v)|}{|C(v)|} > \frac{|CN(v)|}{|N(v)|}$

Table 2. Ratios between cloned and non-cloned code

4.3. Domination of Deletions

As can be seen from the overall results and the detailed discussions for ArgoUML and SQuirreL, the deletions have a very strong influence on the results. To eliminate the effect of deletion peaks, Table 2 shows the number of weeks where the percentage of additions, deletions, or changes to non-cloned code is larger than to cloned code (columns A, C, E) or vice versa (B, D, F). The number is always larger for the non-cloned code.

To smooth out the data, the extreme peak values for deletions of cloned or non-cloned code have been deleted and the average data has been recomputed without the peaks. In each one of the systems, this has eliminated not more than four data sets. The recomputed overall results shown in Figure 12 are now much more clearer and the following conclusions can be drawn:

- The average percentage of additions, deletions, or other changes to clone code is lower than the average percentage for non-cloned code.
- It is more often the case that a higher percentage of non-cloned code is added, deleted, or changed in comparison to cloned code.
- Non-cloned code is more often changed (incl. additions and deletions) than cloned code and therefore, cloned is more stable than non-cloned-code.

4.4. Threats to Validity

There are some potential threats to validity in the presented study. First of all, there is no clear definition of a clone. Moreover, a clone detected by a clone detector may not be a clone in reality (false positive) or a clone in a system may be missed by a clone detector (false negative). To reduce the number of false positives, we have used

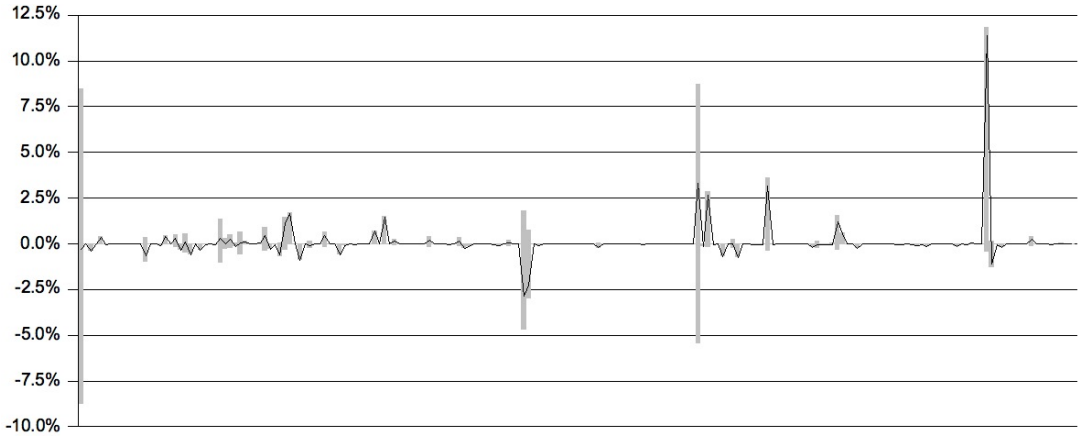

Figure 9. SQuirreL: Deletions

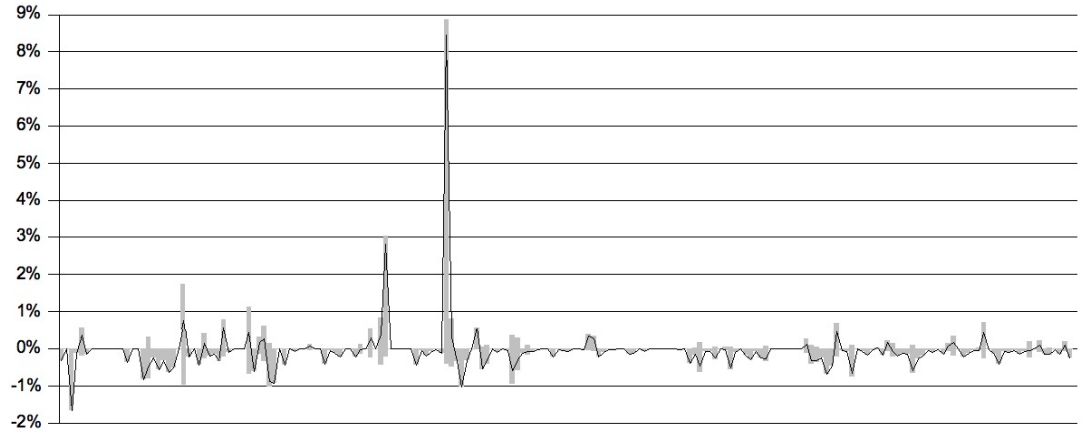

Figure 10. SQuirreL: Additions

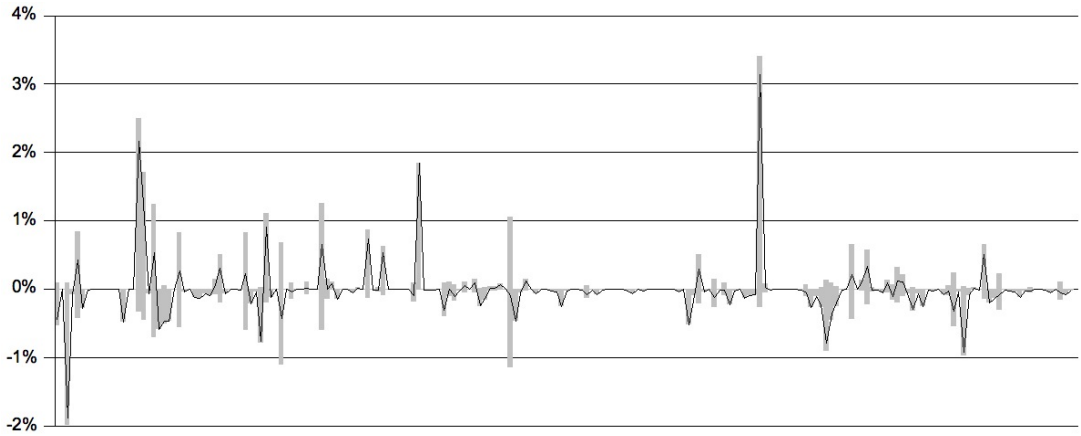

Figure 11. SQuirreL: Changes (that are neither Additions nor Deletions)

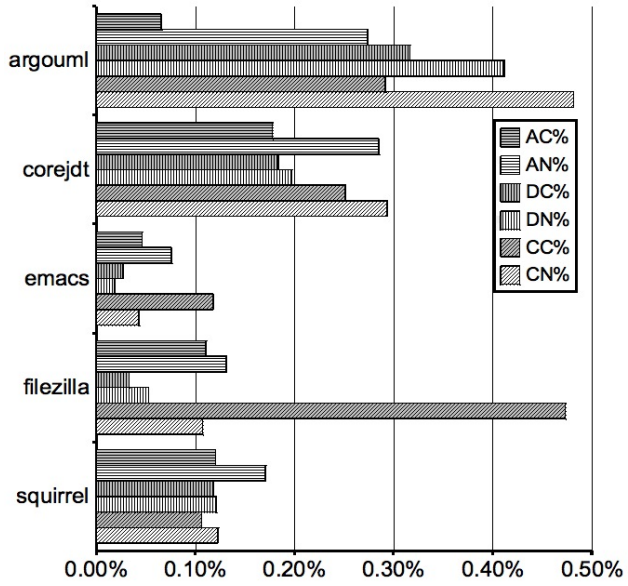

Figure 12. Smoothed Summary of the Results

Simian with strict settings such that only identical clones are detected. In addition, the analyzed systems have been transformed by removing comments and pretty printing. It is known that clone detectors have a low recall [7], so the false negatives cause a threat to validity which cannot be estimated. Another potential threat to validity is caused by the technique to detect changes with *diff*, however, the risk is reduced by transformation of the analyzed systems and by ignoring whitespace in changes. Because *diff* does not identify the movement of text or code, refactorings or restructurings cause deletions and additions.

The experiment is also influenced by the analyzed systems. To be able to draw general conclusions, five systems have been chosen that are of different application types, written in different programming languages, are of sufficient size, and went through enough changes.

5. Related Work

There are only a few empirical studies that analyze the effect of changes on the code clones of a system. Geiger et al. [10] studied the relation of code clones and change couplings (files which are committed at the same time, by the same author, and with the same modification description), but could not find a (strong) relation.

Kim et al. [15] investigated the evolution of code clones and provided a classification for evolving code clones. Their work already showed that during the evolution of the code clones, consistent changes are fewer than anticipated. However, the study analyzed the evolution of two very small systems, DNSJava and CAROL, both written in Java, and both are a similar type of application.

Aversano et al. [4] did a similar empirical study with a slightly refined framework. Similar to Kim et al., they analyze so called co-changes that are changes committed by the same author, with the same notes, and within 200 seconds. They used a Java-only clone detector that compares subtrees in the abstract syntax tree. The analyzed systems were DNSJava and ArgoUML. Although Aversano et al. state "that the majority of clone classes is always maintained consistently", the numbers they present contradict this statement: For ArgoUML, they found that 45% of the clone groups underwent consistent changes.

In the previous study [19] a similar framework and experiment was presented to study the evolution of code clones in respect to consistent and inconsistent changes. That study analyzed the same data as this paper for four of the five systems.

Besides the above mentioned empirical studies, there is some not directly related work that focuses on the evolution of systems and the contained code clones, without looking at the changes: Antoniol et al. [3] have analyzed the cloning evolution in the Linux kernel for 19 releases. They found that the Linux system does not contain a relevant fraction of code duplication. Furthermore, they found that code duplication tends to remain stable across releases. Lagüe et al. [20] have analyzed the amount of clones for different versions of a large telecommunication switching software. An experience in applying time series to cloning ratio prediction was presented by Antoniol et al. [2]. Al-Ekram et al. [1] investigated the code cloning across software systems.

Kim et al. [14] studied why and how programmers introduce code clones into software systems. Lagüe et al. [20] show how software development could benefit from the inclusion of code clone detection tools into the development process. The relation of code clones to the reliability and maintainability of a system has been examined by Monden et al. [22].

Jarzabek and Li [11] found that at least 68% of the Java Buffer library's code was contained in cloned classes or class methods. Close analysis of program situations that led to cloning revealed difficulties in eliminating clones with conventional program design techniques. Kapser and Godfrey [13] list several patterns of cloning that are used in real software systems and argue that clones can be a reasonable design decision.

6. Conclusions and Future Work

This work studied the question if cloned code is more stable than non-cloned code during the evolution of a system. The study analyzed the changes that are applied to

code clones during 200 weeks of evolution of five open source software systems. The study has shown:

- The amount of deletions that occur during the evolution of a system is dominating, i.e. there exist massive deletions of cloned code.
- The average percentage of additions, deletions, or other changes to clone code is lower than the average percentage for non-cloned code.
- It is more often the case that a higher percentage of non-cloned code is added, deleted, or changed in comparison to cloned code.
- Non-cloned code is more often changed (incl. additions and deletions) than cloned code and therefore, cloned code is more stable than non-cloned-code.

Because cloned code is more stable than non-cloned code it cannot be generally assumed that the maintenance of cloned code is more expensive than the maintenance of non-cloned code. However, it has been observed that a system can contain cloned code between similar files and that massive restructurings that delete such clones occur in practice.

Currently, the study is expanded with the analysis of more and larger systems. It is also planned to use other clone detection tools than *Simian* to achieve more general results.

References

[1] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *International Symposium on Empirical Software Engineering*, 2005.

[2] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *International Conference on Software Maintenance (ICSM'01)*, pages 273–280, Nov. 2001.

[3] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, Oct. 2002.

[4] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, 2007.

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM'98)*, pages 368–378, 1998.

[7] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diplomarbeit, Universität Stuttgart, 2002. (In German).

[8] J. Cordy. Comprehending reality – practical barriers to industrial adoption of software maintenance automation. In *11th IEEE International Workshop on Program Comprehension*, pages 196–205, 2003.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance (ICSM'99)*, pages 109–118, 1999.

[10] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, number 3922 in LNCS, pages 411–425. Springer, Mar. 2006.

[11] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution*, 18(4):267–292, 2006.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[13] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *13th Working Conference on Reverse Engineering (WCRE'06)*, pages 19–28, 2006.

[14] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[15] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pages 187–196, 2005.

[16] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Eigth International Static Analysis Symposium (SAS)*, volume 2126 of *LNCS*, 2001.

[17] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Fourth Working Conference on Reverse Engineering*, pages 44–54, 1997.

[18] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[19] J. Krinke. A study of consistent and inconsistent changes to code clones. In *14th Working Conference on Reverse Engineering (WCRE)*, Oct. 2007.

[20] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *International Conference on Software Maintenance (ICSM'97)*, pages 314–321, 1997.

[21] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM'96)*, pages 244–254, 1996.

[22] A. Monden, D. Nakae, T. Kamiya, S. ichi Sato, and K. ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Eighth IEEE International Symposium on Software Metrics (METRICS'02)*, 2002.

[23] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? On fridays. In *International Workshop on Mining Software Repositories (MSR)*, 2005.